Assembly & Shortest Common Superstring

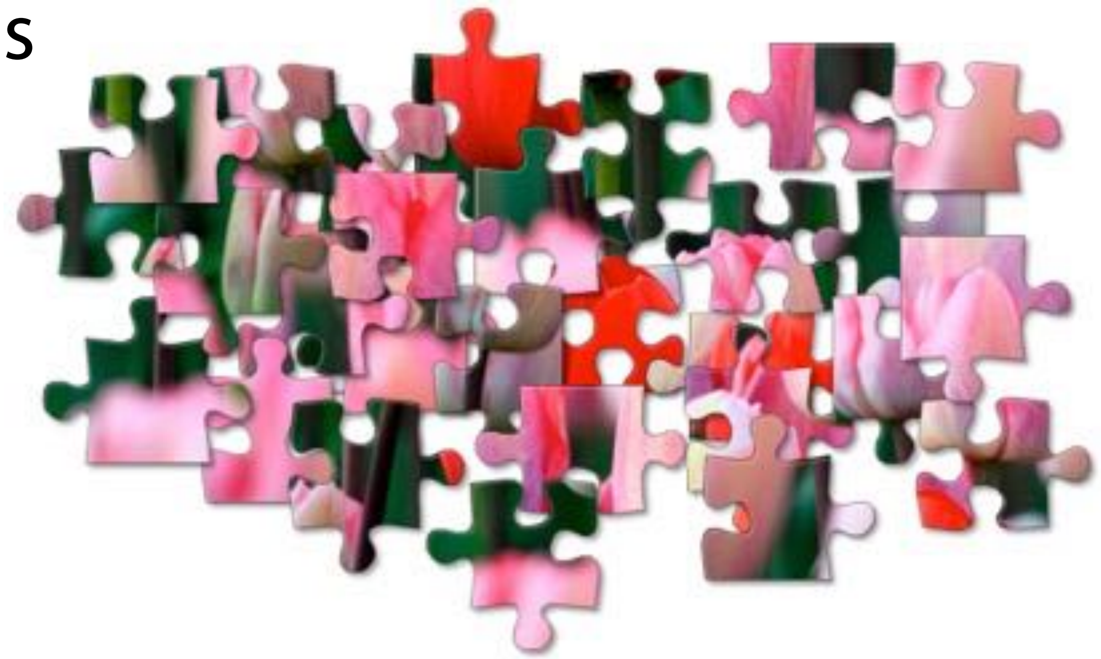
Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Department of Computer Science

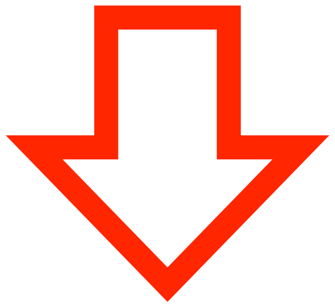
Assembly

Reads



+

Reference genome

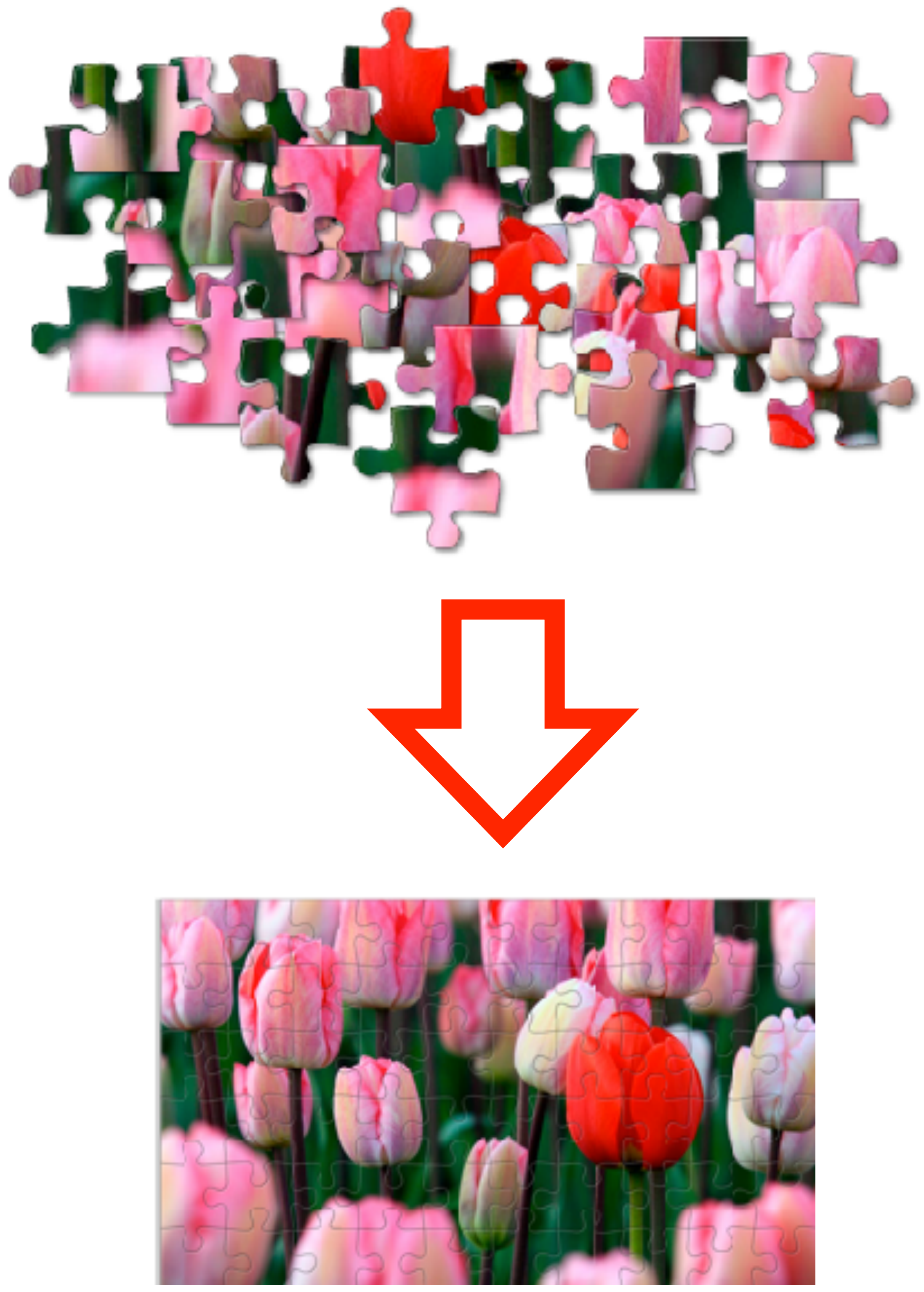


Input DNA

How do we assemble puzzle without the benefit of knowing what the finished product should look like?

(That's what the Human Genome Project had to do!)

De novo shotgun assembly



Assembly

Whole-genome "shotgun" sequencing first copies the input DNA:

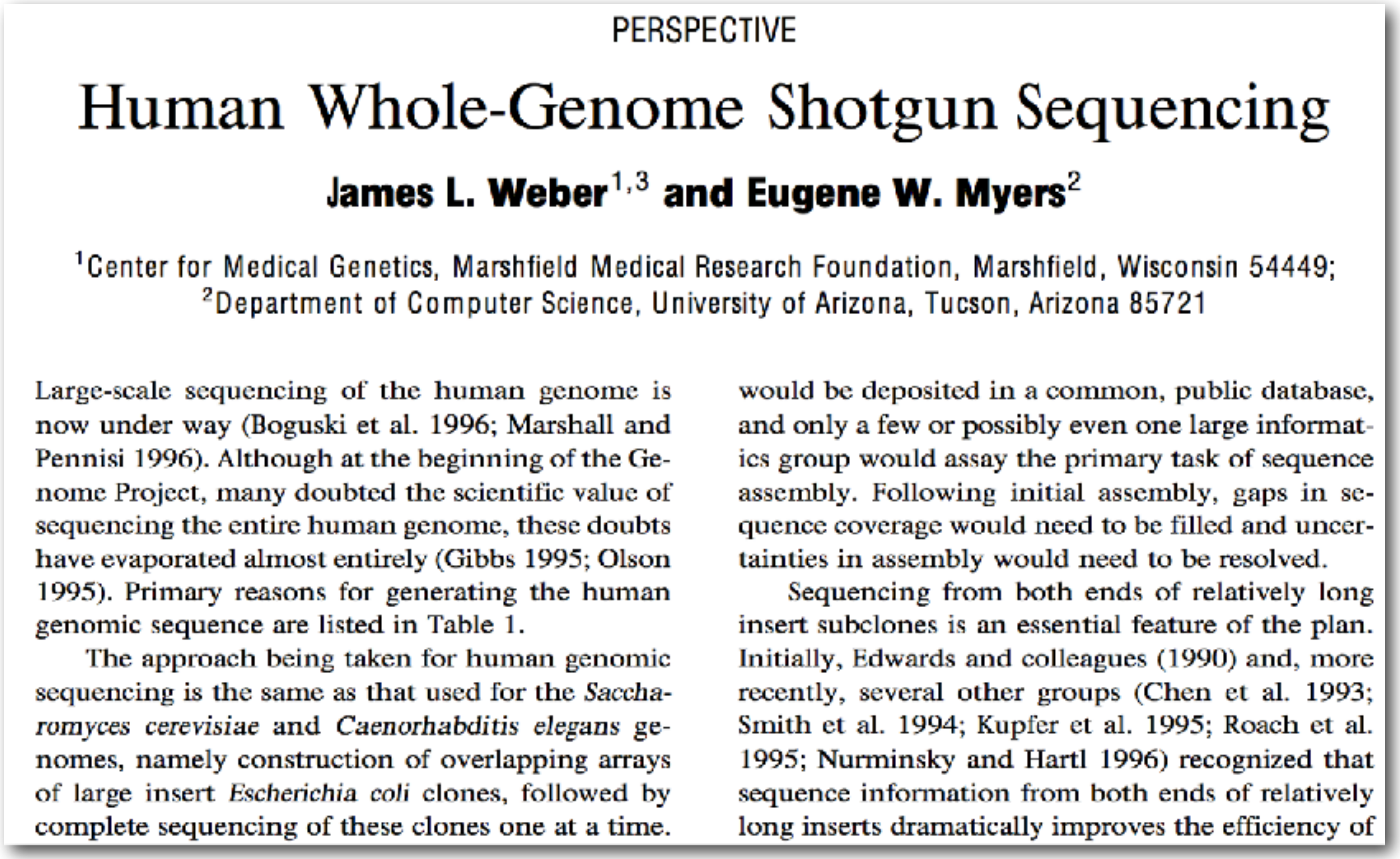
```
Input:  GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Copy:   GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
        GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
        GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
        GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Then fragments it:

```
Fragment: GGCGTCTA   TATCTCGG   CTCTAGGCCCTC   ATTTTTT
          GGC   GTCTATAT   CTCGGCTCTAGGCCCTCA   TTTTTT
          GGCGTC   TATATCT   CGGCTCTAGGCCCT   CATTTTTT
          GGCGTCTAT   ATCTCGGCTCTAG   GCCCTCA   TTTTTT
```

"Shotgun" refers to the random fragmentation of the whole genome; like it was fired from a shotgun

Assembly: Human Genome Project debate



PERSPECTIVE

Human Whole-Genome Shotgun Sequencing

James L. Weber[1,3] and Eugene W. Myers[2]

[1]Center for Medical Genetics, Marshfield Medical Research Foundation, Marshfield, Wisconsin 54449; [2]Department of Computer Science, University of Arizona, Tucson, Arizona 85721

Large-scale sequencing of the human genome is now under way (Boguski et al. 1996; Marshall and Pennisi 1996). Although at the beginning of the Genome Project, many doubted the scientific value of sequencing the entire human genome, these doubts have evaporated almost entirely (Gibbs 1995; Olson 1995). Primary reasons for generating the human genomic sequence are listed in Table 1.

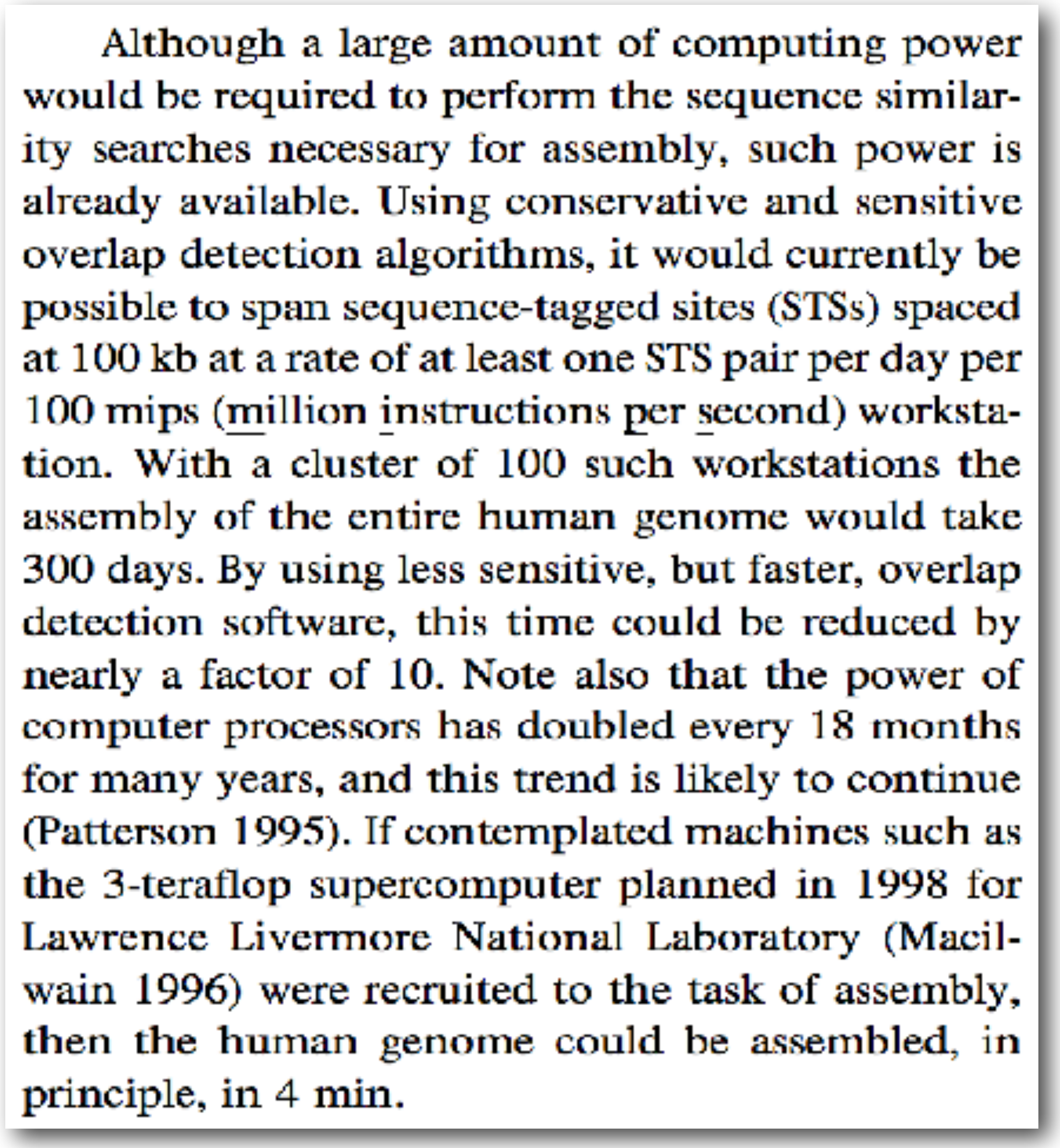
The approach being taken for human genomic sequencing is the same as that used for the *Saccharomyces cerevisiae* and *Caenorhabditis elegans* genomes, namely construction of overlapping arrays of large insert *Escherichia coli* clones, followed by complete sequencing of these clones one at a time.

would be deposited in a common, public database, and only a few or possibly even one large informatics group would assay the primary task of sequence assembly. Following initial assembly, gaps in sequence coverage would need to be filled and uncertainties in assembly would need to be resolved.

Sequencing from both ends of relatively long insert subclones is an essential feature of the plan. Initially, Edwards and colleagues (1990) and, more recently, several other groups (Chen et al. 1993; Smith et al. 1994; Kupfer et al. 1995; Roach et al. 1995; Nurminsky and Hartl 1996) recognized that sequence information from both ends of relatively long inserts dramatically improves the efficiency of

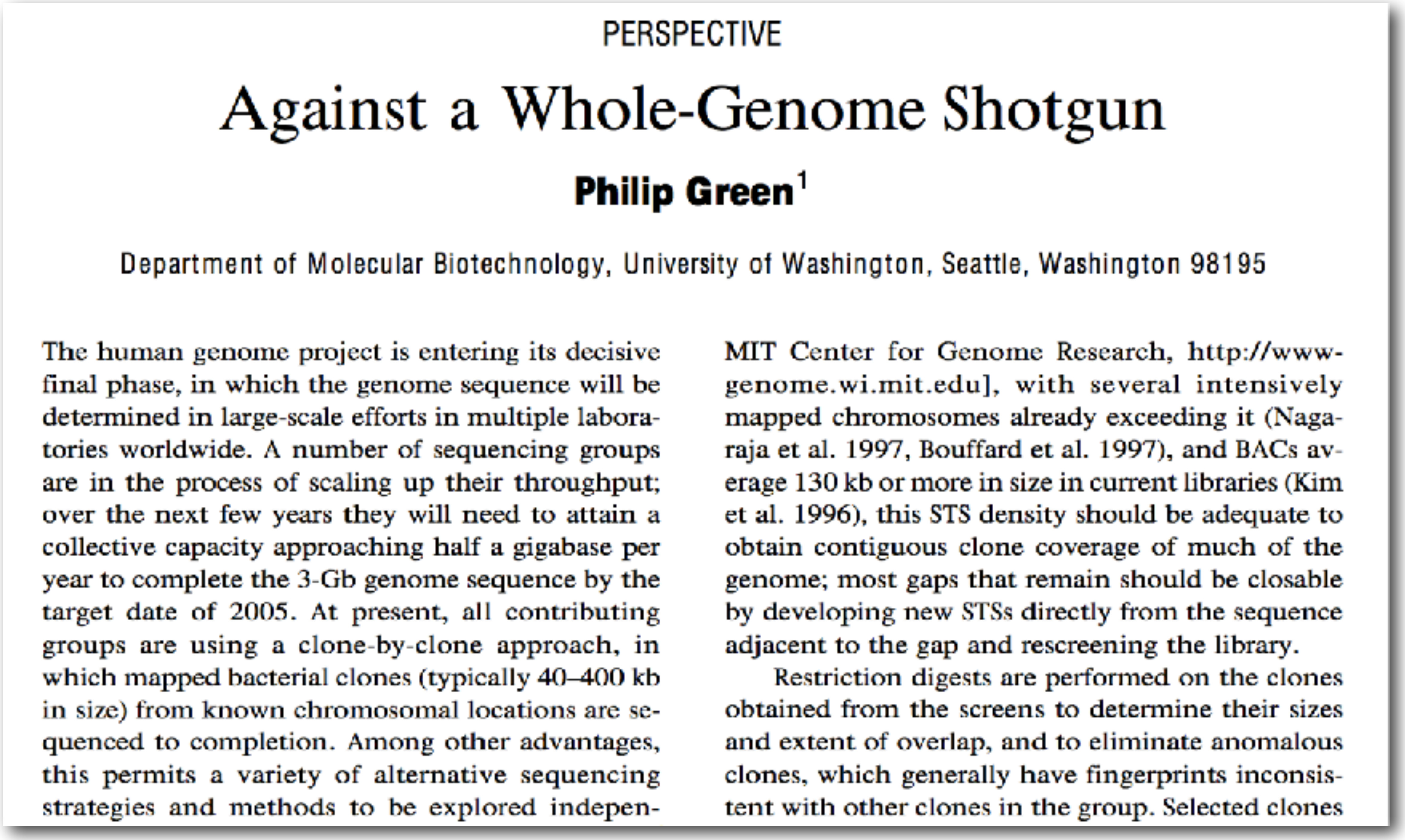
Weber, James L., and Eugene W. Myers. "Human whole-genome shotgun sequencing." *Genome Research* 7.5 (1997): 401-409.

Assembly: Human Genome Project debate


Although a large amount of computing power would be required to perform the sequence similarity searches necessary for assembly, such power is already available. Using conservative and sensitive overlap detection algorithms, it would currently be possible to span sequence-tagged sites (STSs) spaced at 100 kb at a rate of at least one STS pair per day per 100 mips (million instructions per second) workstation. With a cluster of 100 such workstations the assembly of the entire human genome would take 300 days. By using less sensitive, but faster, overlap detection software, this time could be reduced by nearly a factor of 10. Note also that the power of computer processors has doubled every 18 months for many years, and this trend is likely to continue (Patterson 1995). If contemplated machines such as the 3-teraflop supercomputer planned in 1998 for Lawrence Livermore National Laboratory (Macilwain 1996) were recruited to the task of assembly, then the human genome could be assembled, in principle, in 4 min.

Weber, James L., and Eugene W. Myers. "Human whole-genome shotgun sequencing." *Genome Research* 7.5 (1997): 401-409.

Assembly: Human Genome Project debate



PERSPECTIVE

Against a Whole-Genome Shotgun

Philip Green[1]

Department of Molecular Biotechnology, University of Washington, Seattle, Washington 98195

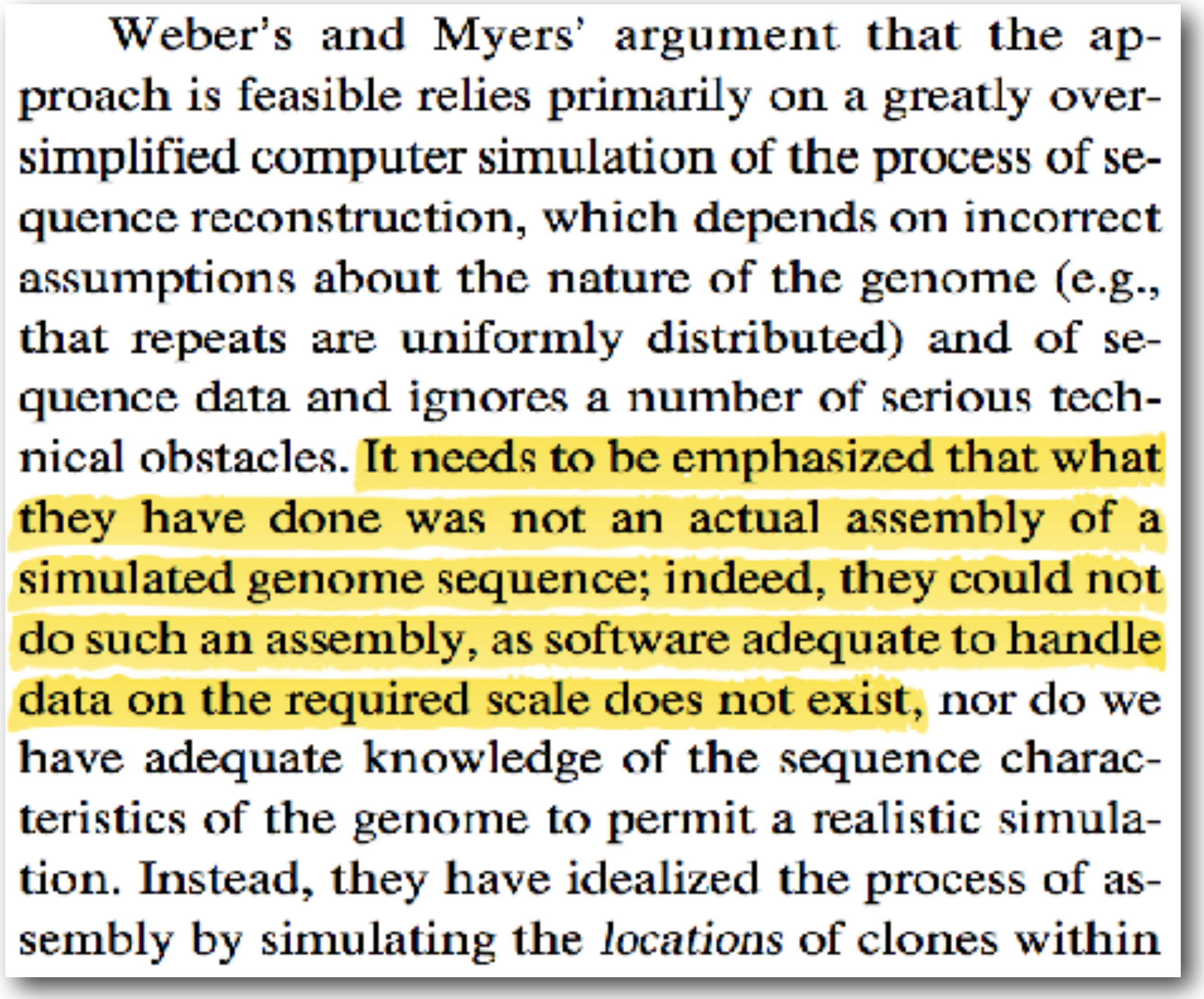
The human genome project is entering its decisive final phase, in which the genome sequence will be determined in large-scale efforts in multiple laboratories worldwide. A number of sequencing groups are in the process of scaling up their throughput; over the next few years they will need to attain a collective capacity approaching half a gigabase per year to complete the 3-Gb genome sequence by the target date of 2005. At present, all contributing groups are using a clone-by-clone approach, in which mapped bacterial clones (typically 40–400 kb in size) from known chromosomal locations are sequenced to completion. Among other advantages, this permits a variety of alternative sequencing strategies and methods to be explored indepen-

MIT Center for Genome Research, http://www-genome.wi.mit.edu], with several intensively mapped chromosomes already exceeding it (Nagaraja et al. 1997, Bouffard et al. 1997), and BACs average 130 kb or more in size in current libraries (Kim et al. 1996), this STS density should be adequate to obtain contiguous clone coverage of much of the genome; most gaps that remain should be closable by developing new STSs directly from the sequence adjacent to the gap and rescreening the library.

Restriction digests are performed on the clones obtained from the screens to determine their sizes and extent of overlap, and to eliminate anomalous clones, which generally have fingerprints inconsistent with other clones in the group. Selected clones

Green, Philip. "Against a whole-genome shotgun." *Genome Research* 7.5 (1997): 410-417.

Assembly: Human Genome Project debate



Weber's and Myers' argument that the approach is feasible relies primarily on a greatly oversimplified computer simulation of the process of sequence reconstruction, which depends on incorrect assumptions about the nature of the genome (e.g., that repeats are uniformly distributed) and of sequence data and ignores a number of serious technical obstacles. It needs to be emphasized that what they have done was not an actual assembly of a simulated genome sequence; indeed, they could not do such an assembly, as software adequate to handle data on the required scale does not exist, nor do we have adequate knowledge of the sequence characteristics of the genome to permit a realistic simulation. Instead, they have idealized the process of assembly by simulating the *locations* of clones within

Green, Philip. "Against a whole-genome shotgun."
Genome Research 7.5 (1997): 410-417.

Assembly

From these

```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
```

Reconstruct this

```
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Assembly

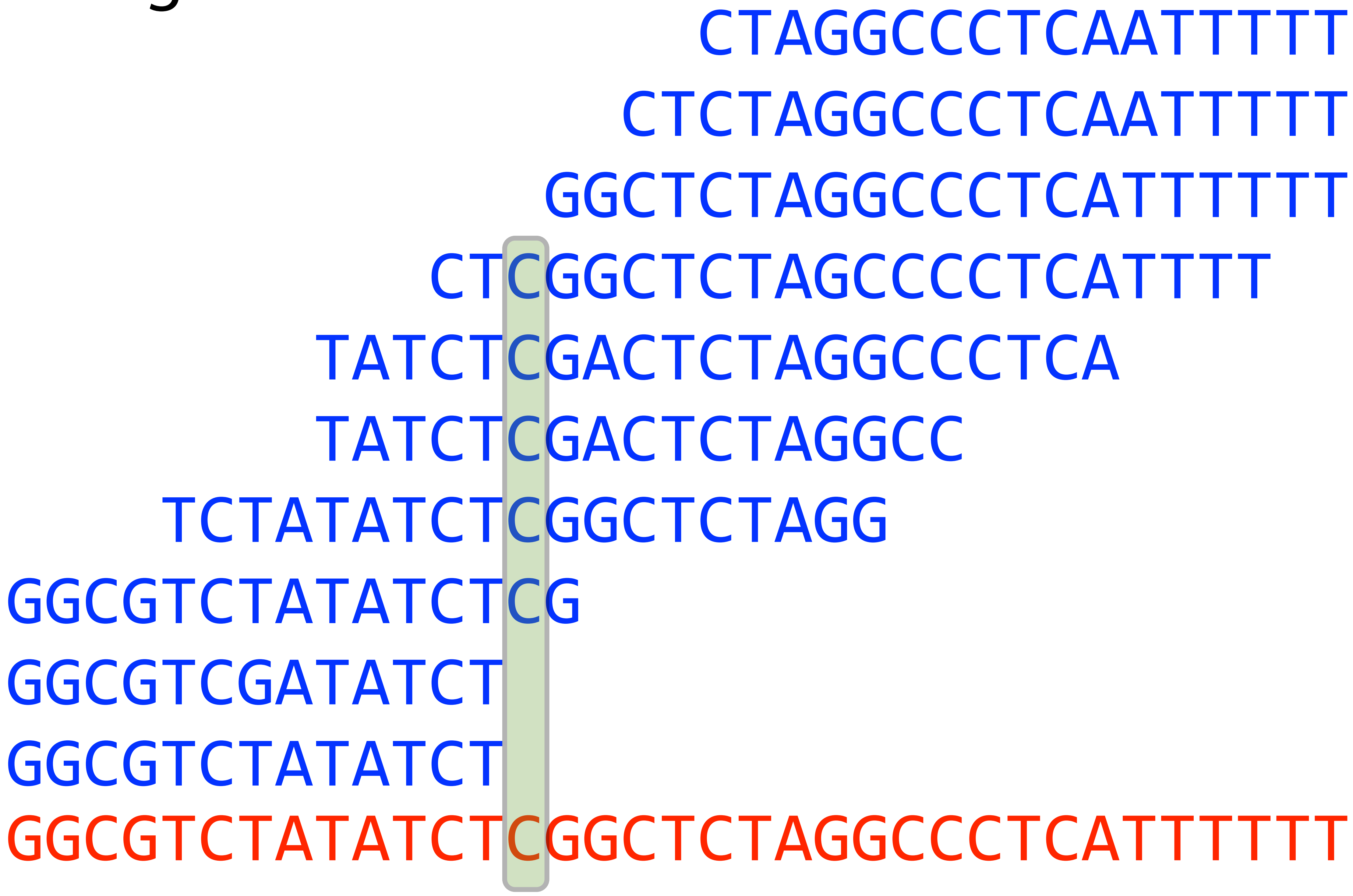
```
CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG
```

From these

Reconstruct this

???????????????????????????????????

Coverage



```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Coverage = 5

Coverage

```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Coverage = 5

```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC               177 bases
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT                            35 bases
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Average coverage = 177 / 35 ≈ 5-fold

```
TCTATATCTCGGCTCTAGG
    TATCTCGACTCTAGGCC
```

```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
    TATCTCGACTCTAGGCC
```

First law of assembly

If a suffix of read A is similar to a prefix of read B...

```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
    TATCTCGACTCTAGGCC
```

...then A and B might *overlap* in the genome

```
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
        TATCTCGACTCTAGGCC
```

```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
    TATCTCGACTCTAGGCC
           ↑
```

Why the differences?

1. Sequencing errors
2. Ploidy: e.g. humans have 2 copies of each chromosome, and copies can differ

Second law of assembly

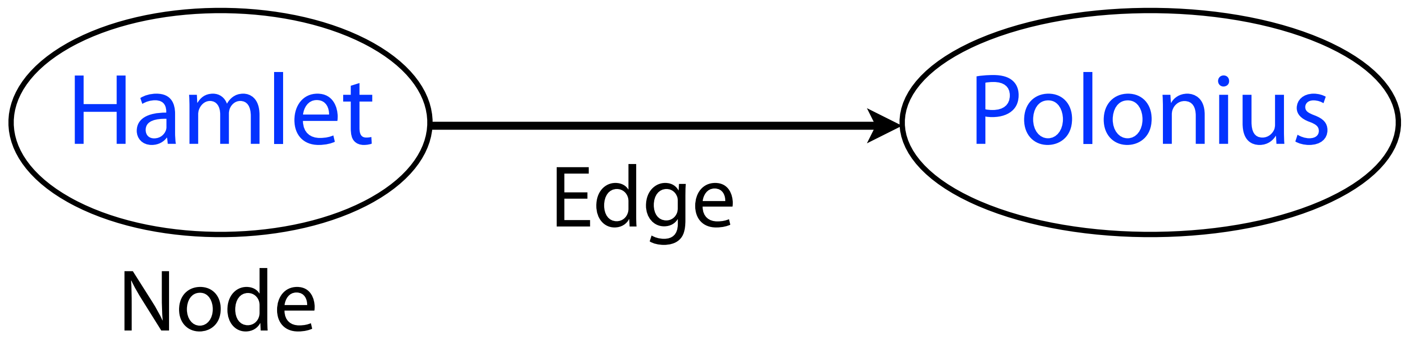
More coverage leads to more and longer overlaps

```
                  CTAGGCCCTCAATTTTT
           CTCGGCTCTAGCCCCTCATTTT
    TCTATATCTCGGCTCTAGG                 less coverage
GGCGTCGATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
                  CTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCTATATCT                          more coverage
```

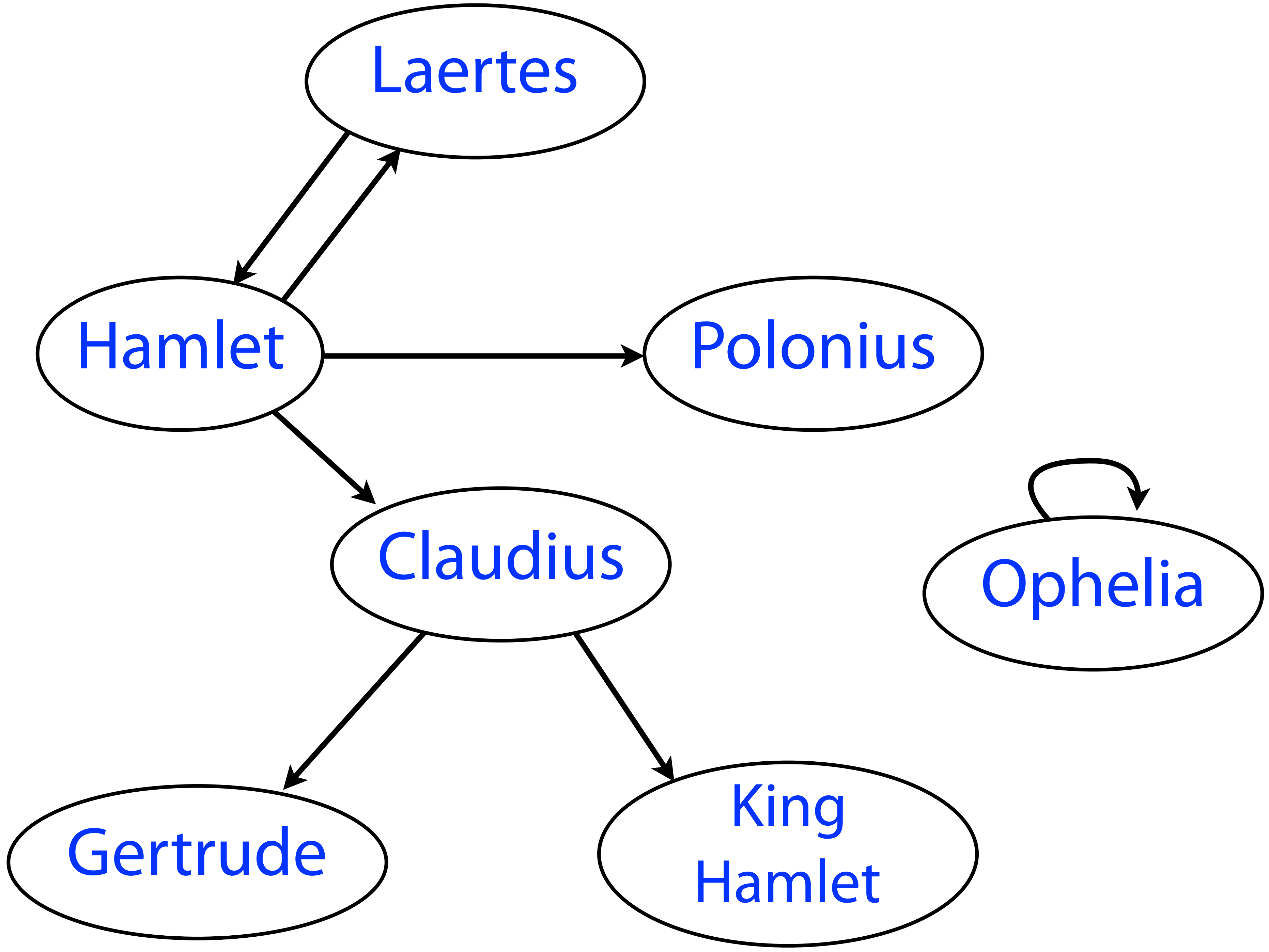
```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
    TATCTCGACTCTAGGCC
```

```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
    TATCTCGACTCTAGGCC

    TATCTCGACTCTAGGCC
       |||| |||||| ||
       CTCGGCTCTAGCCCCTCAT
```

Directed graph



Directed graph

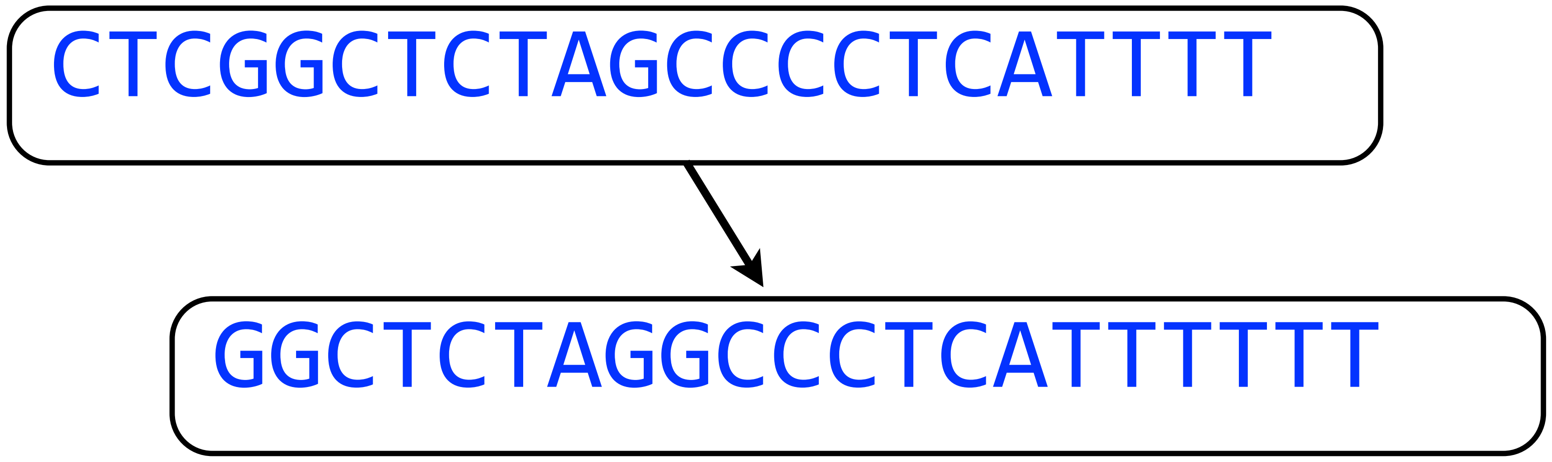


Overlap graph

Each node is a read

CTCGGCTCTAGCCCCTCATTTT

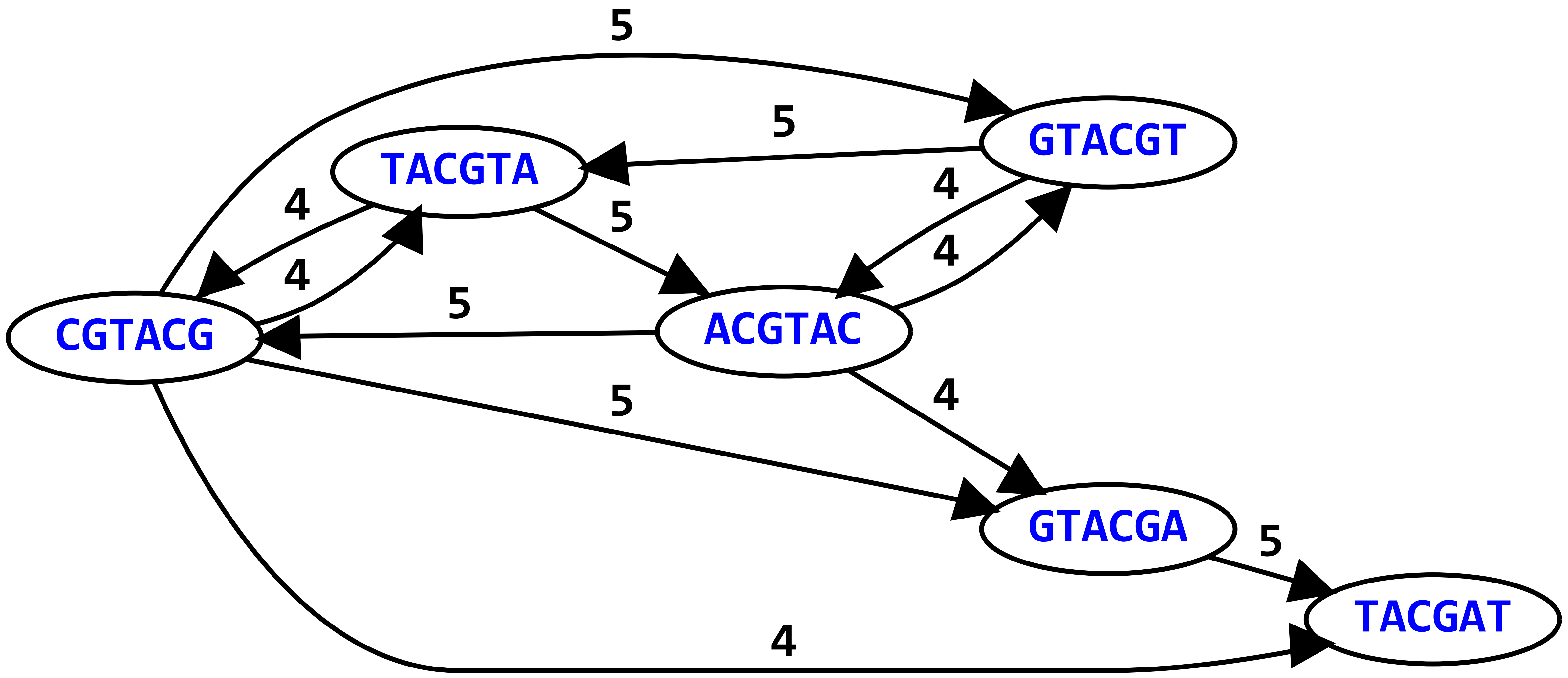
Draw edge A -> B when suffix of A overlaps prefix of B



Overlap graph

Nodes: all 6-mers from GTACGTACGAT

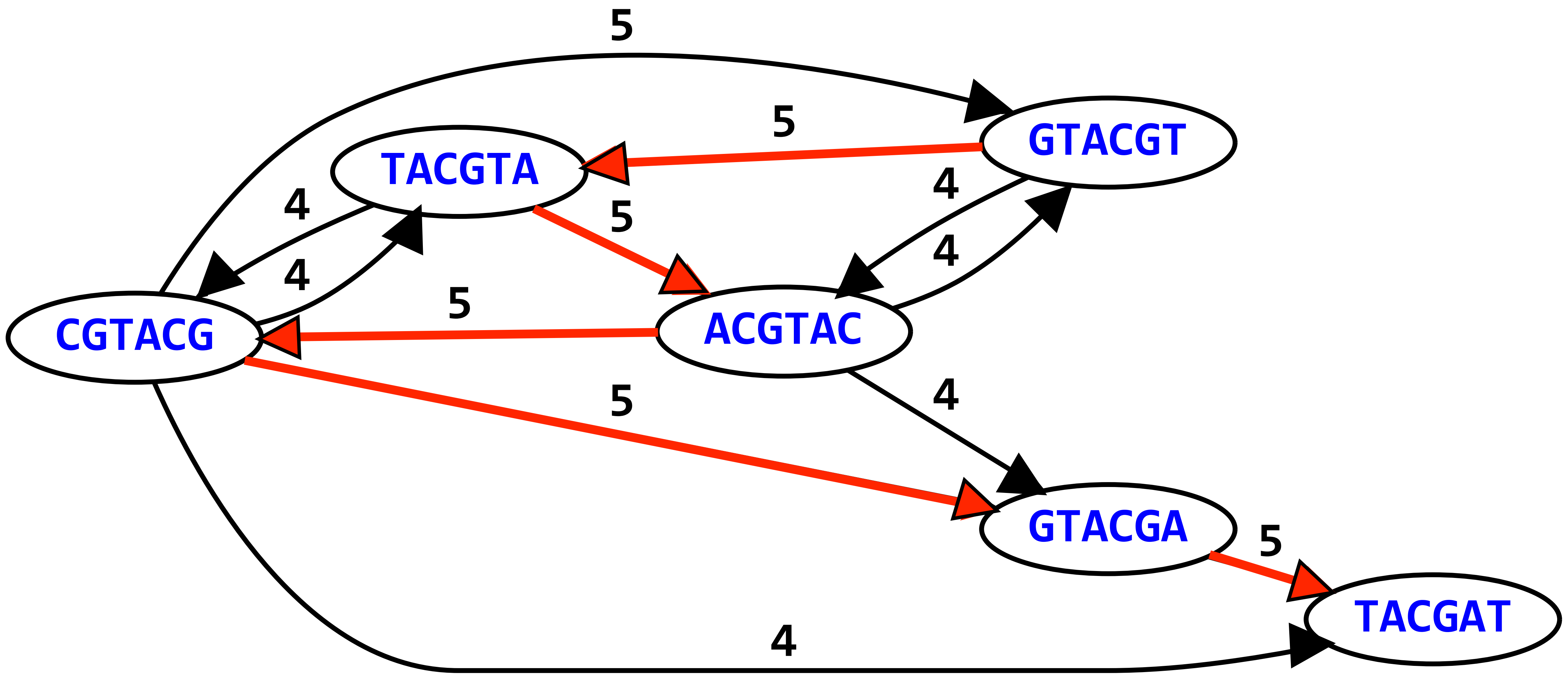
Edges: overlaps of length ≥4



Overlap graph

Nodes: all 6-mers from GTACGTACGAT

Edges: overlaps of length ≥4



Shortest common superstring

Given set of strings *S*, find *SCS*(*S*): shortest string containing the strings in *S* as substrings

S: BAA AAB BBA ABA ABB BBB AAA BAB

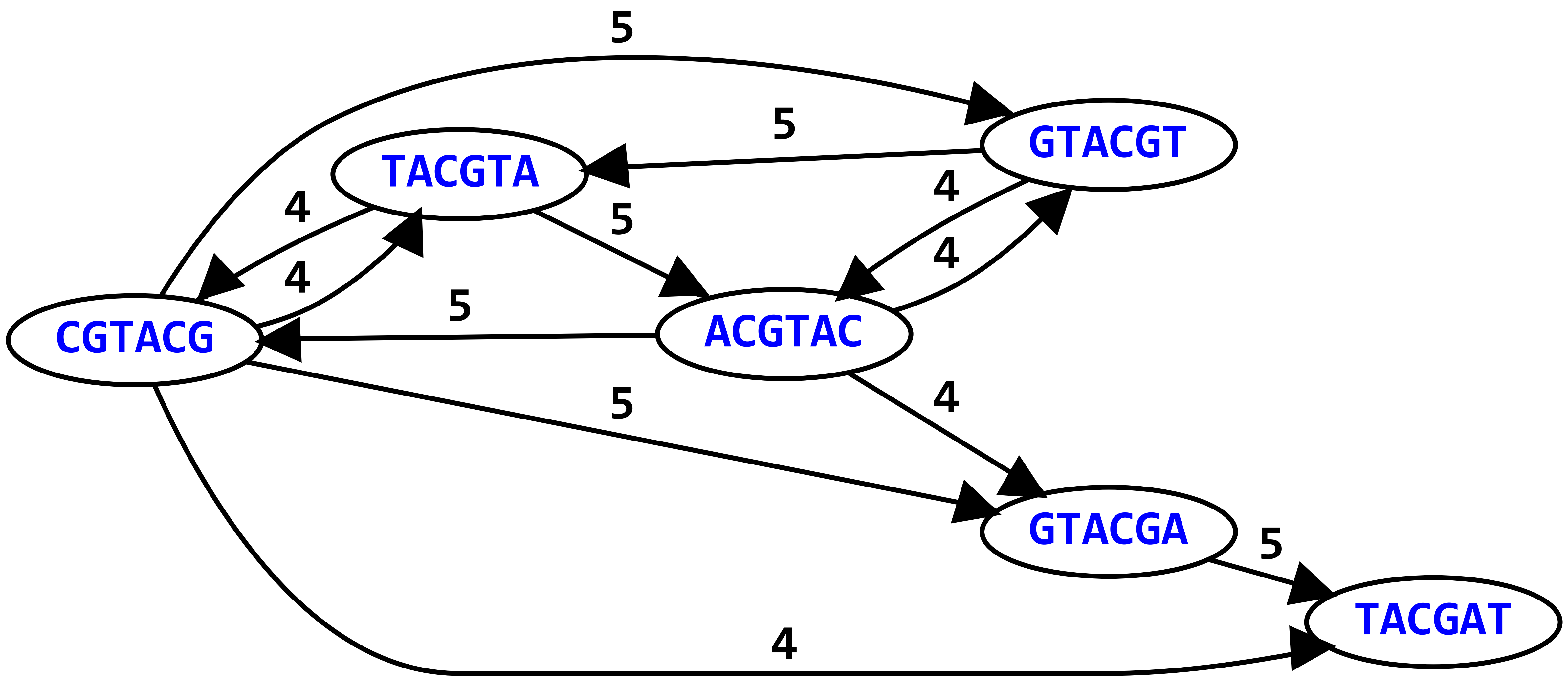
Concat(*S*): BAAAABBBAABAABBBBBAAABAB

$\longmapsto$ 24 $\longleftarrow$

SCS(*S*): AAABBBABAA

$\longmapsto$ 10 $\longleftarrow$

Reads: all 6-mers from GTACGTACGAT



```
>>> scs(['GTACGT', 'TACGTA', 'ACGTAC',
         'CGTACG', 'GTACGA', 'TACGAT'])
'GTACGTACGAT'
```

Shortest common superstring

NP-complete: no efficient algorithms for large inputs

Idea: pick order for strings in *S and* construct superstring

order 1: $\underbrace{\text{AAA AAB}}$ ABA ABB BAA BAB BBA BBB

AAA

Idea: pick order for strings in *S* *and* construct superstring

order 1: $\underbrace{\text{AAA}\quad\text{AAB}}$ ABA ABB BAA BAB BBA BBB

AAAB

Idea: pick order for strings in *S and* construct superstring

order 1: AAA $\underline{\text{AAB ABA}}$ ABB BAA BAB BBA BBB

AAABA

Idea: pick order for strings in *S and* construct superstring

order 1: AAA AAB $\underbrace{\text{ABA ABB}}$ BAA BAB BBA BBB

AAABABB

Idea: pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB $\longleftarrow$ superstring 1

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB ⟵ superstring 1

order 2: AAA AAB ABA BAB ABB BBB BAA BBA

AAABABBBAABBA ⟵ superstring 2

Try all possible orderings and pick shortest superstring

If S contains *n* strings, *n*! (*n* factorial) orderings possible *order 1*: AAA AAB ABA ABB BAA BAB BBA BBB

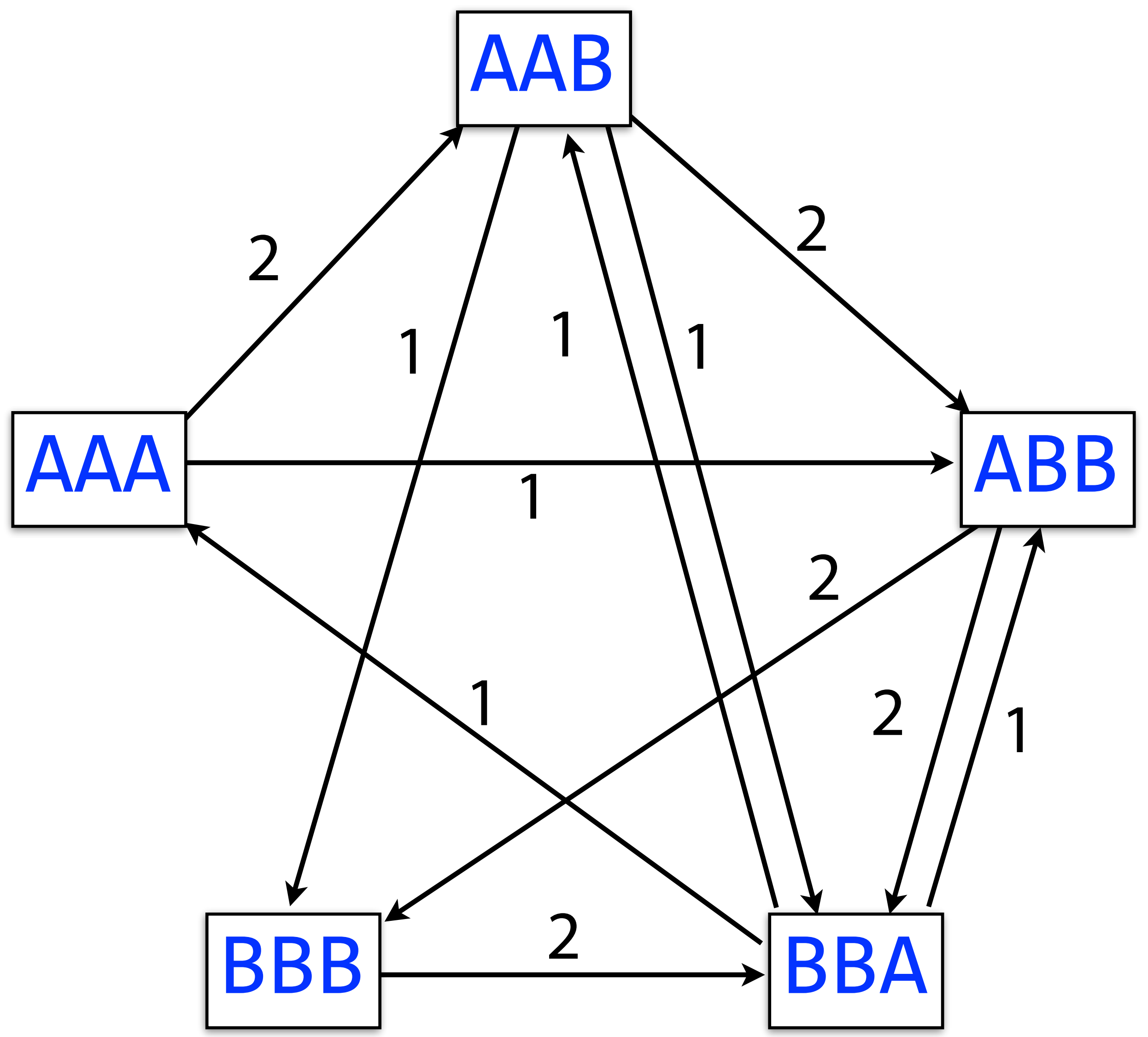
AAABABBAABABBABBB $\longleftarrow$ superstring 1

order 2: AAA AAB ABA BAB ABB BBB BAA BBA

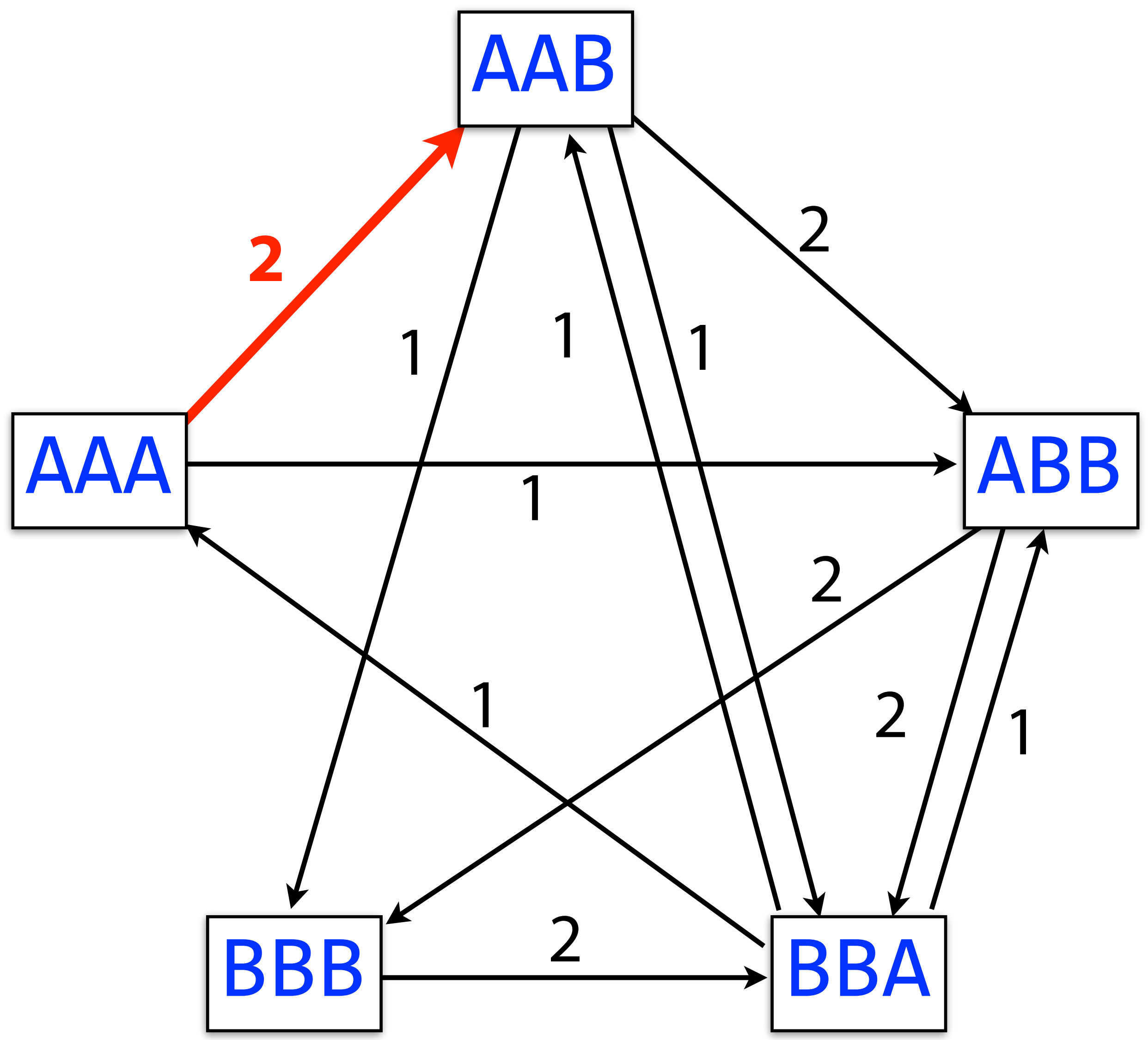
AAABABBBAABBA $\longleftarrow$ superstring 2

If *S* contains *n* strings, *n* ! (*n* factorial) orderings possible

Greedy shortest common superstring



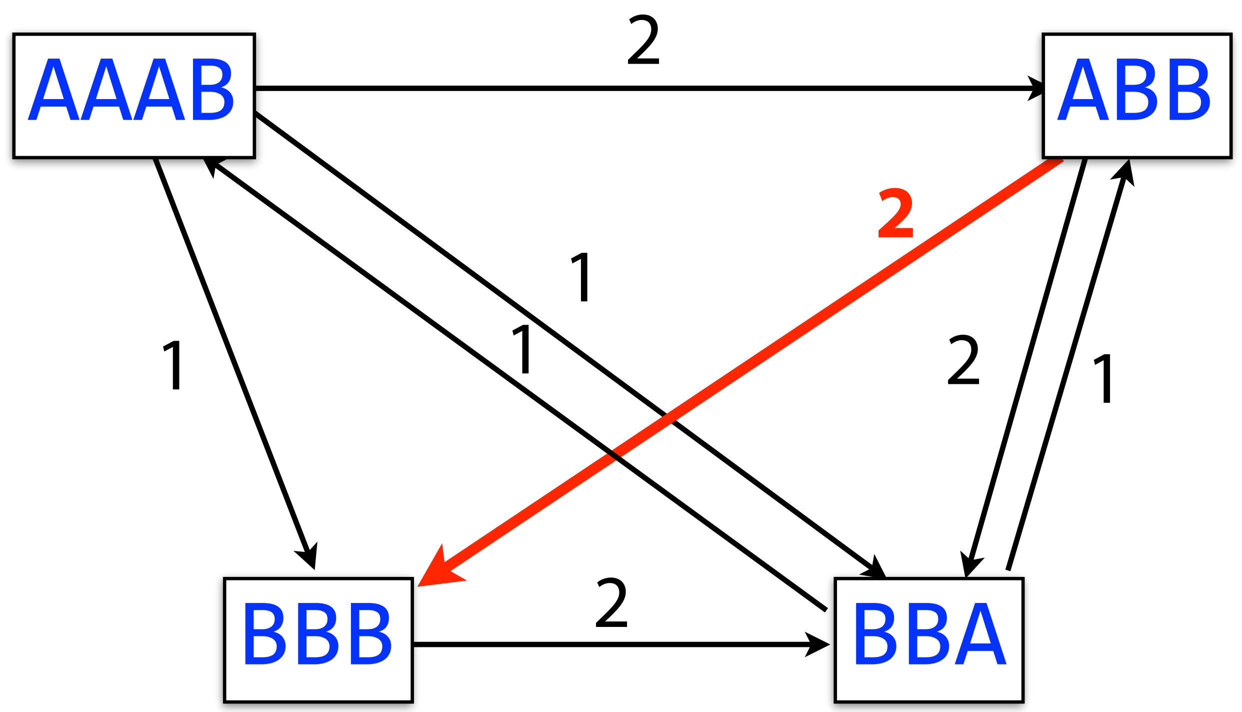
Greedy shortest common superstring



Greedy shortest common superstring

$$\begin{tikzcd}
AAAB \arrow[rr, "2"] \arrow[d, "1"'] \arrow[drr, bend left=10, "1"] & & ABB \arrow[dll, "2"'] \arrow[d, bend right=10, "2"'] \\
BBB \arrow[rr, "2"] & & BBA \arrow[ull, bend left=10, "1"] \arrow[u, bend right=10, "1"']
\end{tikzcd}$$

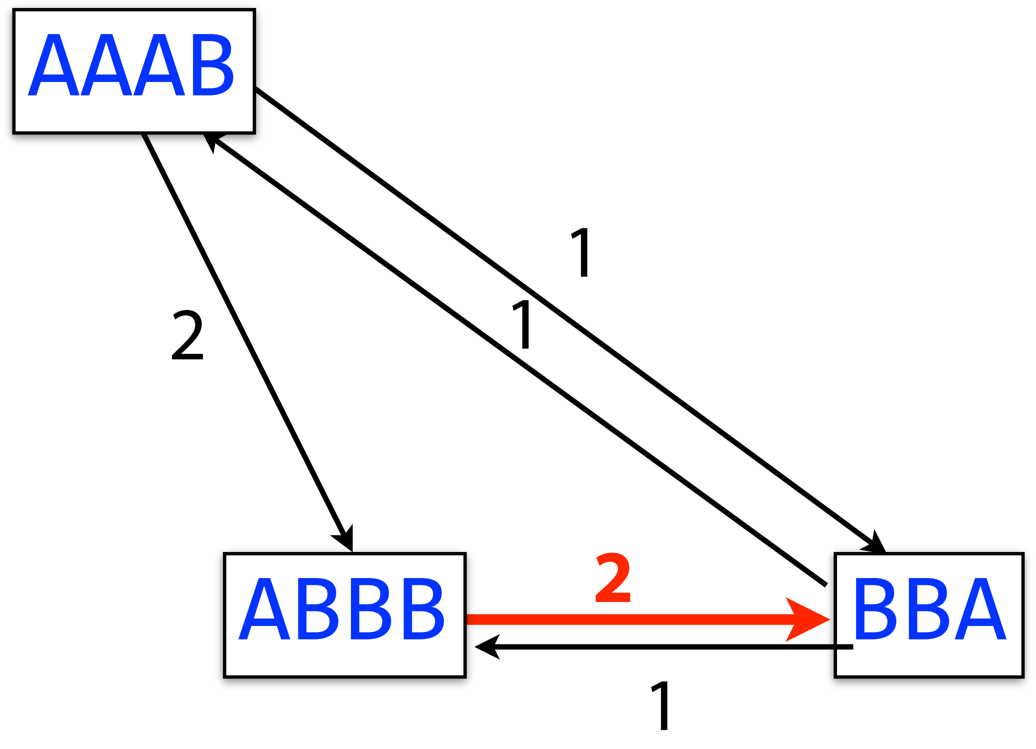
Greedy shortest common superstring



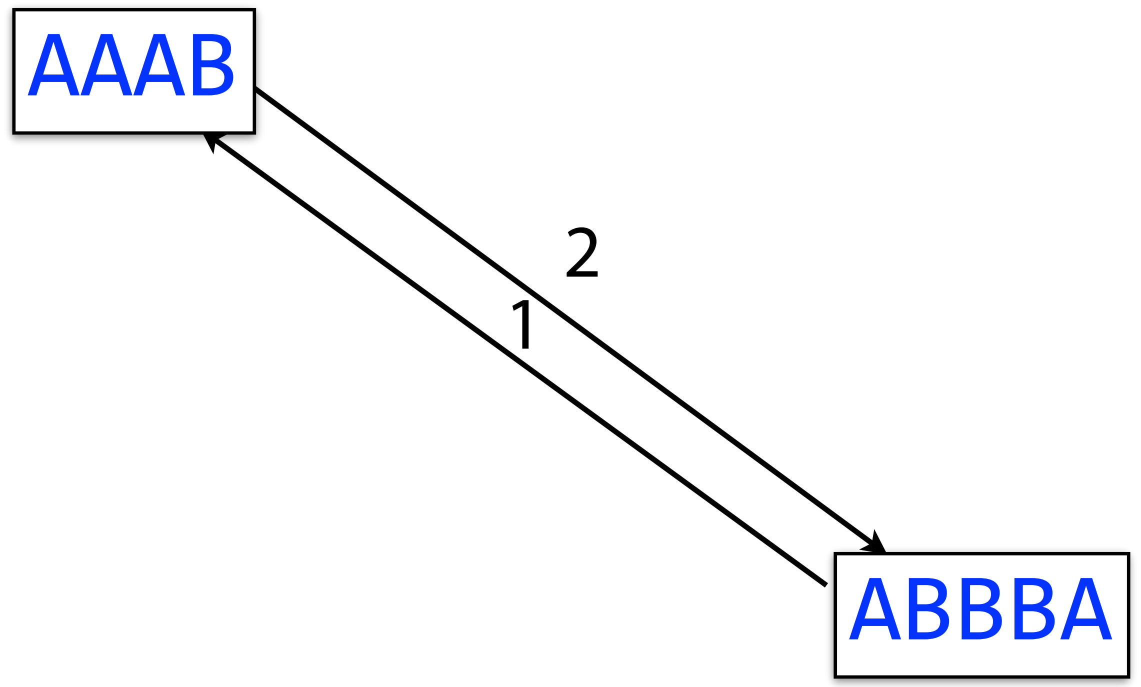
Greedy shortest common superstring

$$\begin{tikzcd}
\boxed{AAAB} \arrow[d, "2"'] \arrow[dr, shift left, "1"] & \\
\boxed{ABBB} \arrow[r, shift left, "2"] & \boxed{BBA} \arrow[ul, shift left, "1"] \arrow[l, shift left, "1"]
\end{tikzcd}$$

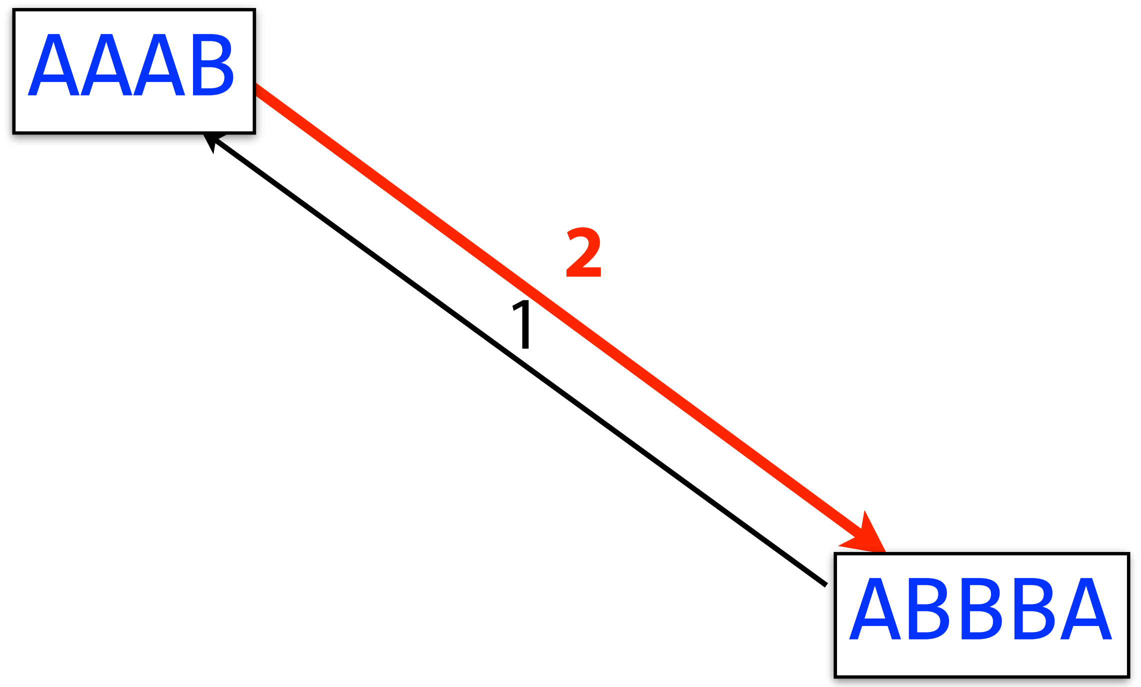
Greedy shortest common superstring



Greedy shortest common superstring



Greedy shortest common superstring


$$\begin{tikzcd}
\boxed{AAAB} \arrow[r, bend left, "2"] & \boxed{ABBBA} \arrow[l, bend left, "1"]
\end{tikzcd}$$

Greedy shortest common superstring

$\boxed{\text{AAABBBA}} \longleftarrow$ superstring, length=7

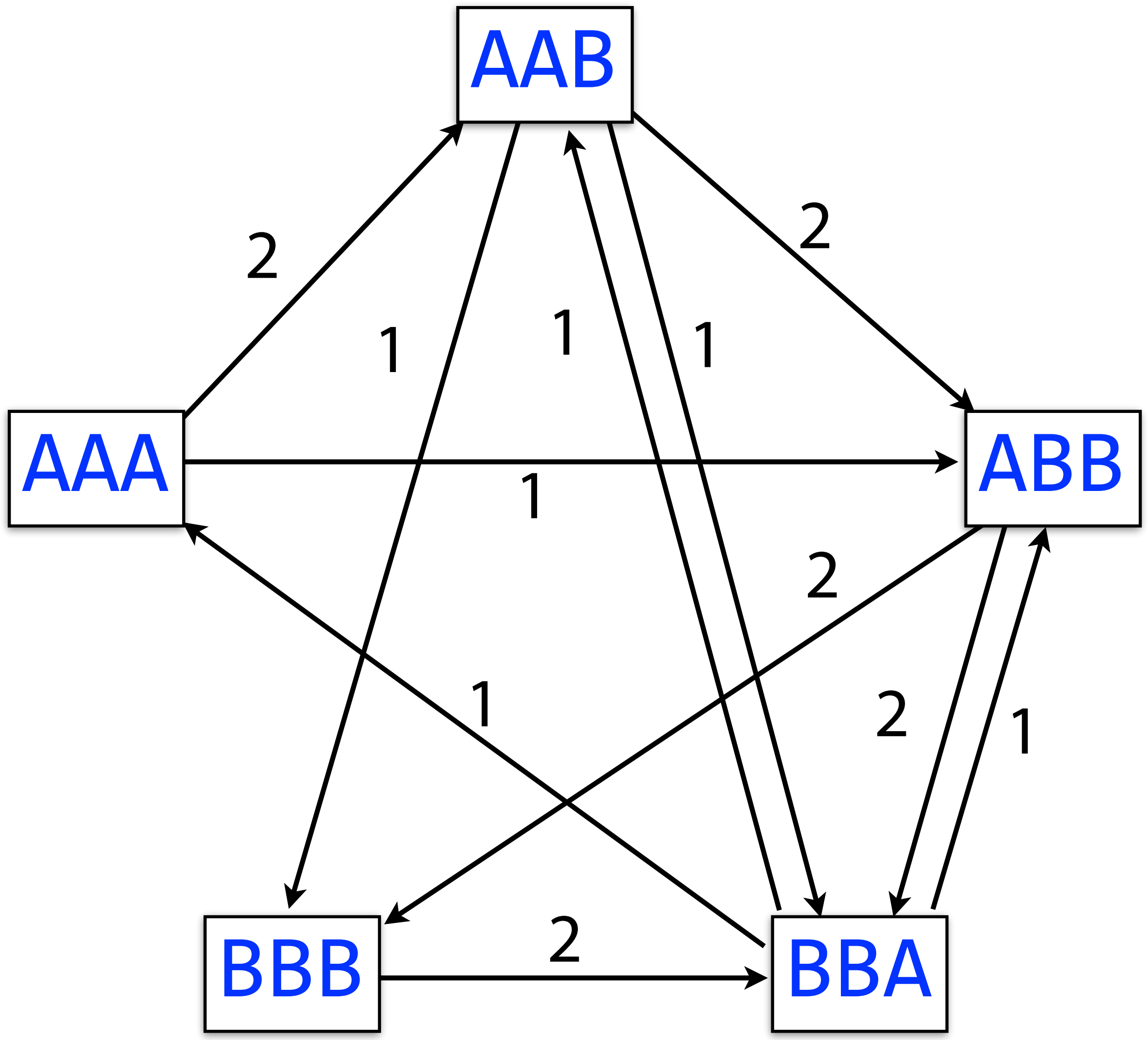
Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

Input strings

AAA AAB ABB BBB BBA



Shortest common superstring: greedy

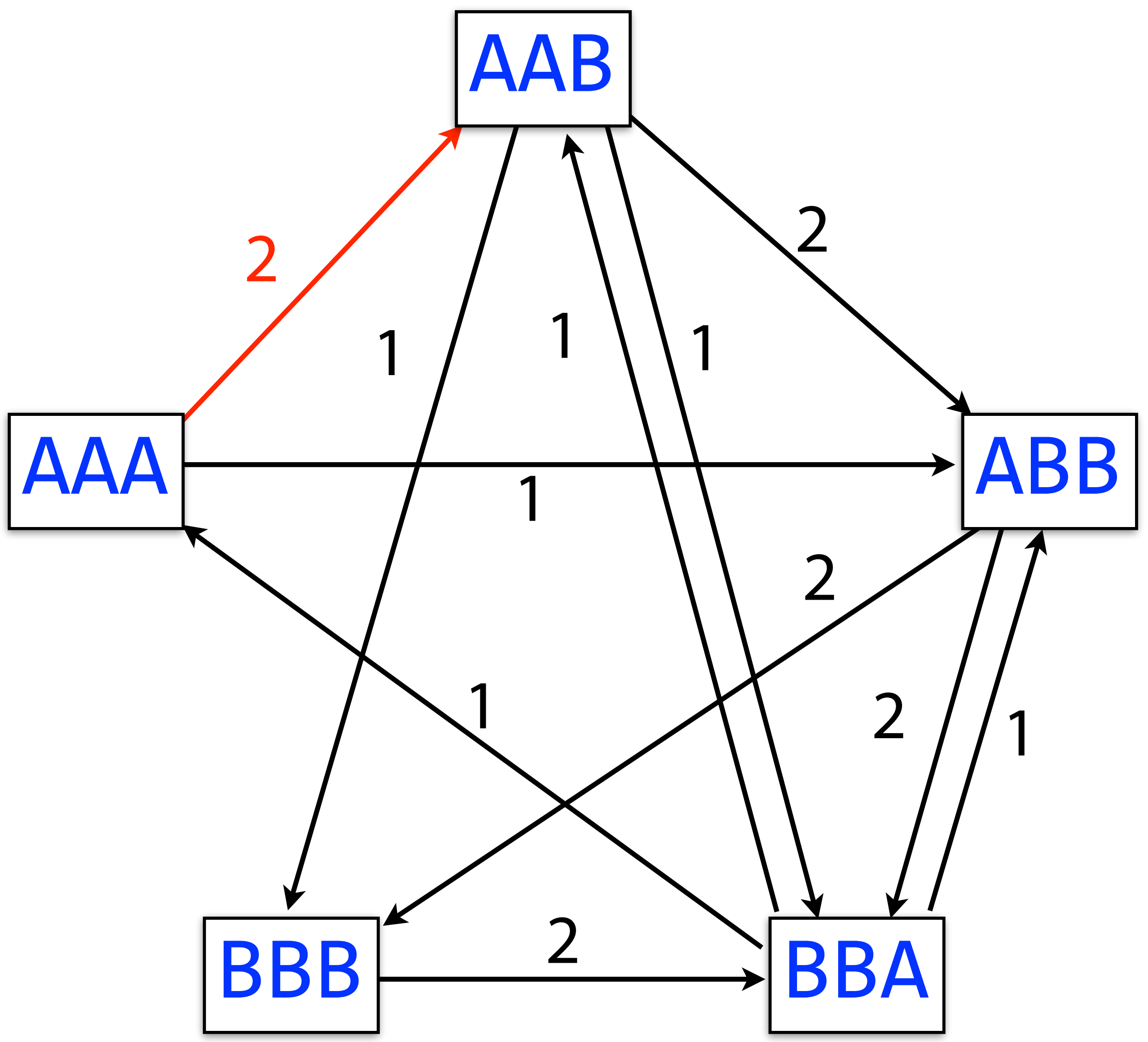
Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

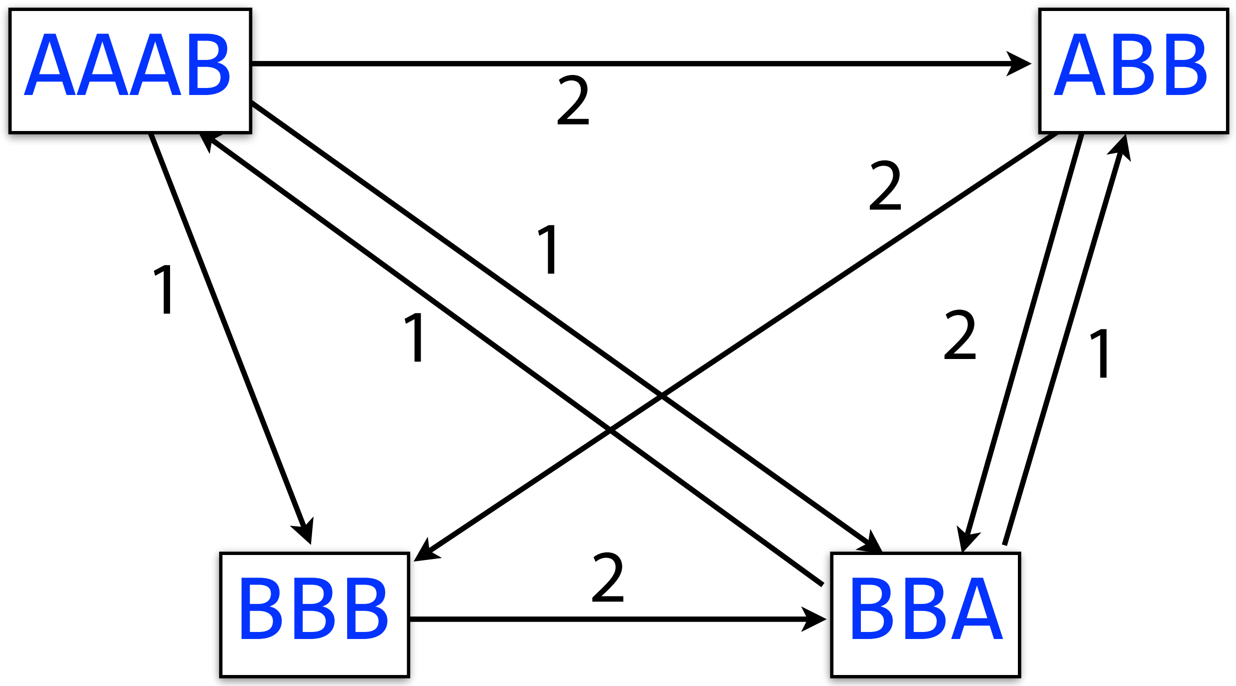
Algorithm in action ($l = 1$):

|——— Input strings ———|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

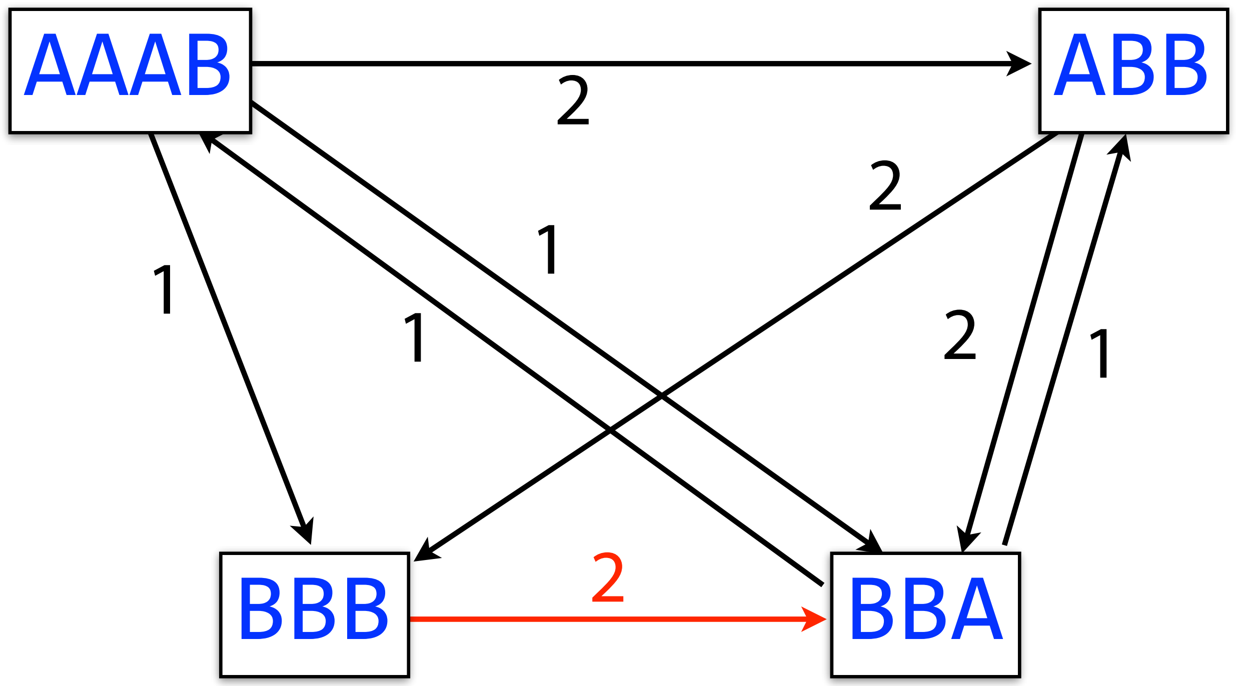
Algorithm in action ($l = 1$):

|——— Input strings ———|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



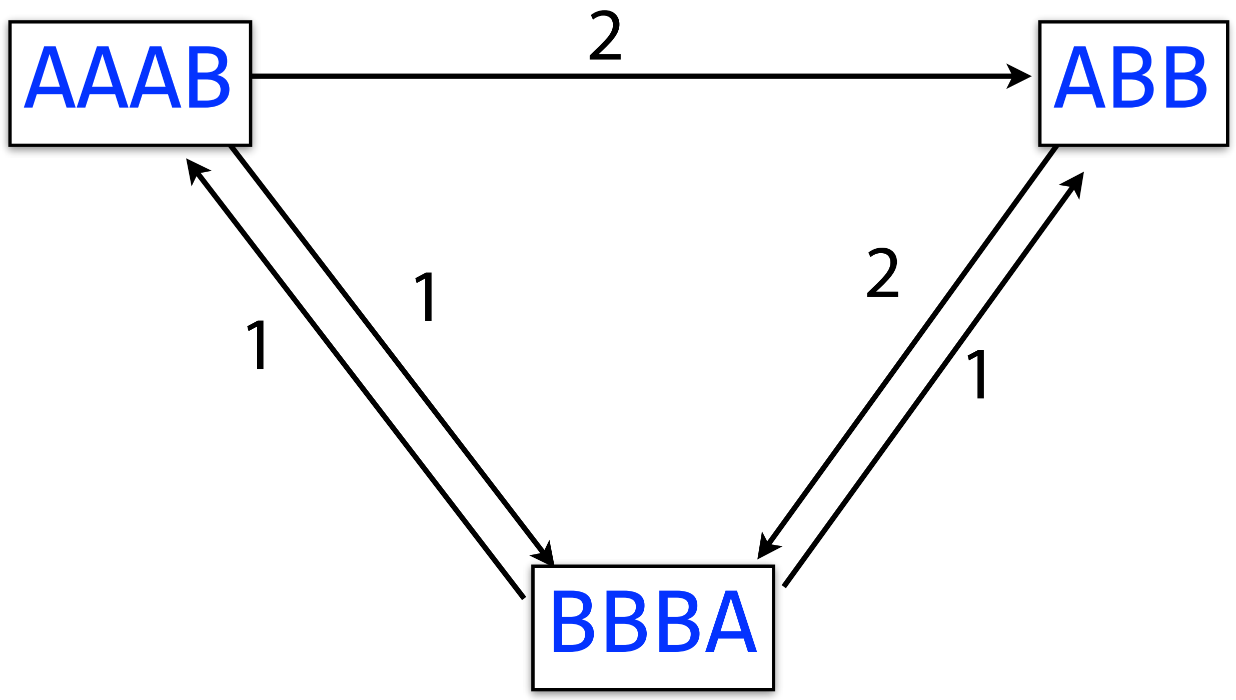
Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA	AAB	ABB	BBB	BBA
AAA	AAB	ABB	BBB	BBA
AAAB	ABB	BBB	BBA	
AAAB	BBBA	ABB		



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action ($l = 1$):

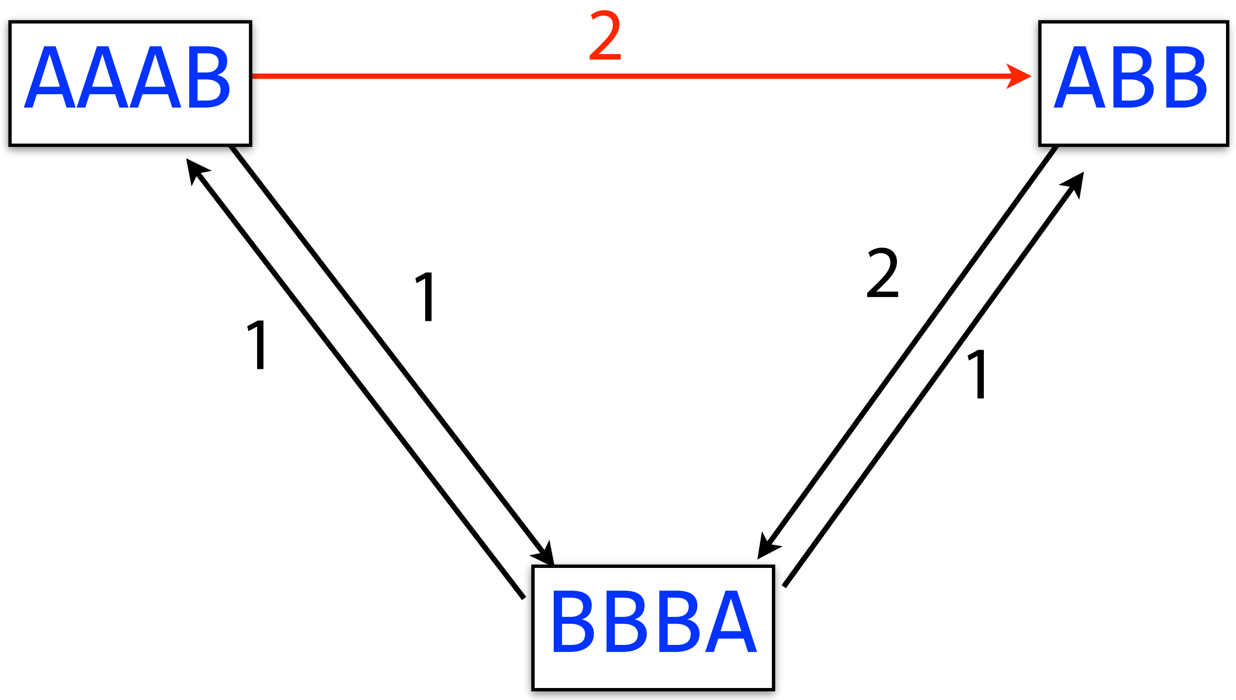
|—— Input strings ——|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action ($l = 1$):

├── Input strings ──┤

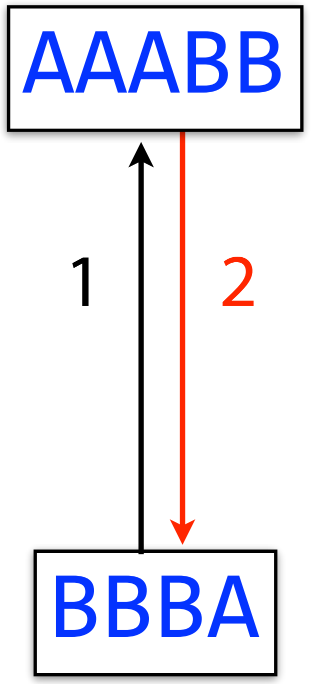
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action ($l = 1$):

|——Input strings——|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA

AAABBBA

$\boxed{\text{AAABBBA}}$

That's the SCS

Greedy shortest common superstring

$$\begin{tikzcd}
AAA \arrow[d] & AAB \arrow[dl] & ABB & BBA & BBB \\
AAAB & ABB & BBA & BBB &
\end{tikzcd}$$

Greedy shortest common superstring

$$\begin{tikzcd}
AAA \arrow[d] & AAB \arrow[dl] & ABB & BBA & BBB \\
AAAB & ABB \arrow[d] & BBA \arrow[dl] & BBB & \\
AAAB & ABBA & BBB & &
\end{tikzcd}$$

Greedy shortest common superstring

$$\begin{tikzcd}
AAA \arrow[d] & AAB \arrow[dl] & ABB & BBA & BBB \\
AAAB & ABB \arrow[d] & BBA \arrow[dl] & BBB & \\
AAAB \arrow[d] & ABBA \arrow[dl] & BBB & & \\
AAABBA & BBB & & &
\end{tikzcd}$$

Greedy shortest common superstring

$$\begin{tikzcd}
\text{AAA} \arrow[d] & \text{AAB} \arrow[dl] & \text{ABB} & \text{BBA} & \text{BBB} \\
\text{AAAB} & \text{ABB} \arrow[d] & \text{BBA} \arrow[dl] & \text{BBB} & \\
\text{AAAB} \arrow[d] & \text{ABBA} \arrow[dl] & \text{BBB} & & \\
\text{AAABBA} \arrow[d] & \text{BBB} \arrow[dl] & & & \\
\text{AAABBABBB} & \text{superstring, length=9} \arrow[l] & & &
\end{tikzcd}$$

Greedy shortest common superstring

$$\begin{tikzcd}
AAA \arrow[d] & AAB \arrow[dl] & ABB & BBA & BBB \\
AAAB & & ABB \arrow[d] & BBA \arrow[dl] & BBB \\
AAAB \arrow[d] & & ABBA \arrow[dll] & & BBB \\
AAABBA \arrow[d] & & & & BBB \arrow[dllll] \\
AAABBABBB & \arrow[l] \text{superstring, length=9}
\end{tikzcd}$$

$$\begin{tikzcd}
AAABBBA & \arrow[l] \text{superstring, length=7}
\end{tikzcd}$$

Greedy answer *isn't necessarily optimal*

Shortest common superstring: greedy

Greedy-SCS assembling all substrings of length 6 from:
`a_long_long_long_time`. $l = 3$.

```
ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
ng_time ong_lon long_ti g_long_ a_long long_l
ong_lon long_time g_long_ a_long long_l
long_lon long_time g_long_ a_long
long_lon g_long_time a_long
long_long_time a_long
a_long_long_time
```

↑

Foiled by repeat!

Shortest common superstring: greedy

Same example, but increased the substring length from 6 to 8

```
long_lon ng_long_ _long_lo g_long_t ong_long g_long_l ong_time a_long_l _long_ti long_tim
long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l _long_ti
_long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l
_long_time a_long_lo long_lon ng_long_ g_long_t ong_long g_long_l
_long_time ong_long_ a_long_lo long_lon g_long_t g_long_l
g_long_time ong_long_ a_long_lo long_lon g_long_l
g_long_time ong_long_ a_long_lon g_long_l
g_long_time ong_long_l a_long_lon
g_long_time a_long_long_l
a_long_long_long_time
a_long_long_long_time
```

Got the whole thing: `a_long_long_long_time`

Shortest common superstring: greedy

Why are substrings of length 8 long enough for Greedy-SCS to figure out there are 3 copies of `long`?

```
a_long_long_long_time
    g_long_l
```

One length-8 substring spans all three `long`s

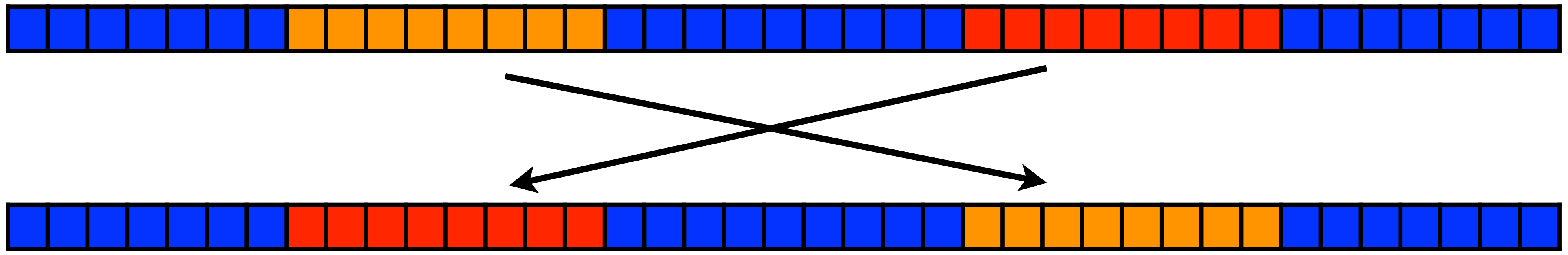
Third law of assembly

Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome

Collapsing a *tandem* repeat:

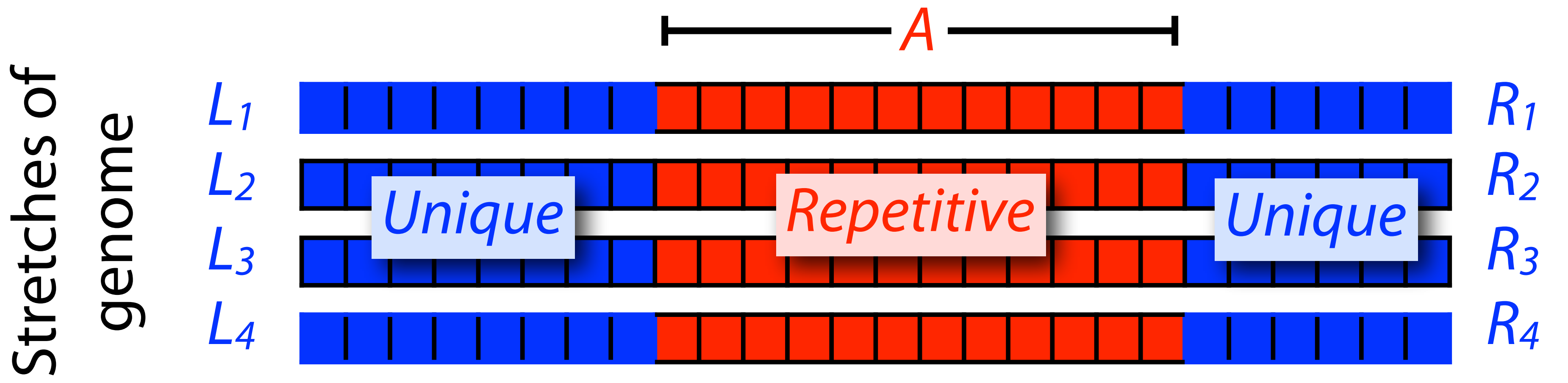
```
a_long_long_long_time
          ↓
   a_long_long_time
```

Spurious rearrangement:

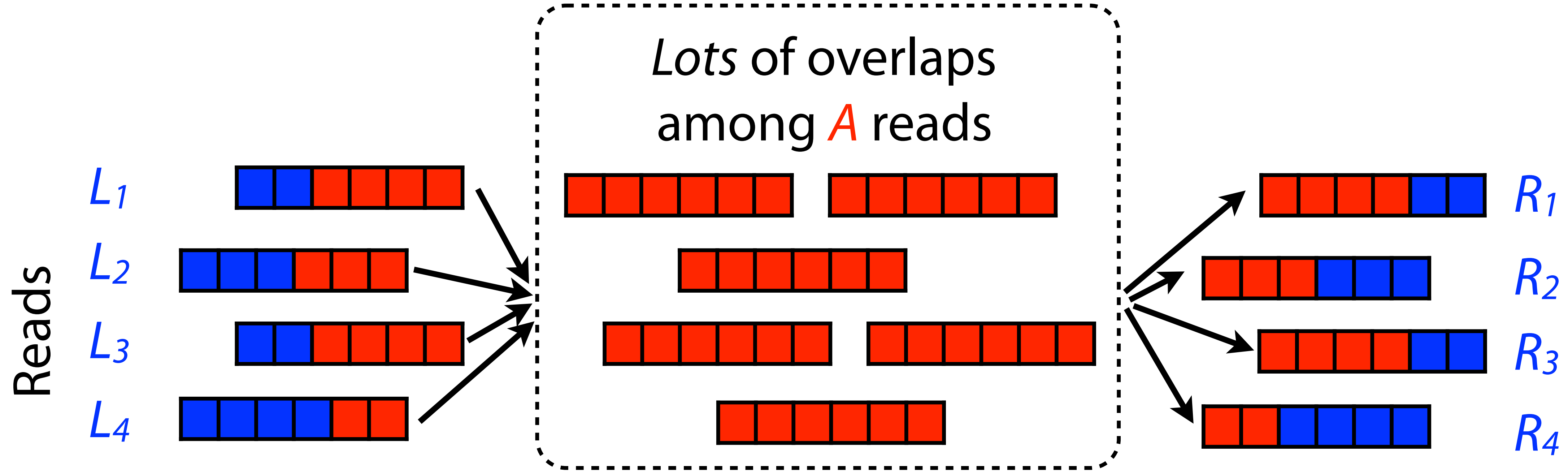


Repeats foil assembly

Portion of overlap graph involving repeat family *A*



*A*s are longer than read length



Even if we avoid collapsing copies of *A*, we can't know which paths *in* correspond to which paths *out*